CBSE and Embedded Systems – Do They Match?

Dominik Auf der Maur, Otto Preiss[1]
ABB Corporate Research Ltd, Switzerland
{dominik.auf-der-maur ¦ otto.preiss}@ch.abb.com

Thomas Siegrist[2]
ABB Business Services Ltd, Switzerland
thomas.siegrist@ch.abb.com

Alain Wegmann
École Polytechnique Fédérale de Lausanne, Switzerland
alain.wegmann@epfl.ch

Abstract

This position paper explicitly aims to facilitate a lively discussion on the usefulness of component-based approaches to building software for embedded systems.

We first structure the various promises that component-based software engineering (CBSE) is said to deliver. This is done by elaborating on the different points of view for the concept of software components, which in turn influence the definition and subsequently the pros and cons. After a general characterization of embedded systems we then present a case study representing the current practice in embedded systems' software development within the control system domain. Particular emphasis is put on the encountered problems. Analyzing CBSE promises and their appropriateness to solving the stated problems shall help to assess the potential of CBSE for embedded system software. But it shall also (maybe with the help of the workshop) generate a catalogue with open questions and inadequacies that justify a combined effort by the research community, and lead to new (?) items on their research agenda.

Introduction

The recent trend to increased use of commercial-off-the-shelf (COTS) hardware and software on the one hand, and the increased interconnection and system integration requirements on the other hand, have profound effect on the development of embedded systems. What once seemed to be a niche market with its own rules, also with respect to the reluctant adoption of main software engineering paradigms (e.g., object orientation), has recently attracted main software players such as Sun or Microsoft. Evidence is given by the ongoing efforts in the industrial process automation field to use Windows CE, real-time versions of Windows NT and Java Virtual Machines in the embedded controller area. In addition, the number of process control solutions is growing which use CORBA or COM/DCOM as interconnection technologies. Furthermore, announcements like IBM's VisualAge Micro Edition (http://www.embedded.oti.com/) show the serious attempts of main players to support the development of embedded systems software. As a result, discussions were started early on using CBSE approaches in general, and on using mainstream component software technologies in particular. The spark of component based development [1] did not spare the embedded systems community, although the promise of building better software in less time is rather generic and was on the banner of almost every significant advance in software development over the last decades.

If only one property characterizing the breed of embedded systems could be mentioned, it would be "resource constraints". A firm grip on the scarce resources (CPU, memory, mass storage, HMI) was always instrumental in building software for embedded systems and dominated the software designs and also the design tool environment. But resource constraints and explicit control over resources is not exactly what the makers of component run-time infrastructure or the advocates of layered architectures had in mind. This should, however, not demoralize further efforts to combine CBSE and embedded

---

[1] Corresponding author

[2] The author worked with ABB Power Automation AG and ABB Automation Products AB at the time of the project described as case study in this paper.

systems. After all, object orientation, too, had a long way to go until it was accepted in the embedded systems software world, but as the case study proofed, clearly brought major improvements.

Taking all the above into consideration, it is legitimate to ask whether the recent advances in CBSE and the respective technologies justify a paradigm shift in developing software for embedded systems. We try to tackle this question by presenting a case study where the current state-of-the-practice in the (non-CBSE) software development of an industrial controller for electrical substation control is described. Our main objective is to identify the "real" problems that embedded software developers and subsequently their organizations are currently facing. Comparing the CBSE state-of-the-art and its potential promises with the identified problems shall allow a more rational discussion on the value of switching to CBSE. Furthermore, it shall reveal the possible shortcomings and stimulate and motivate a research agenda.

For a component related paper these days, it is inevitable to state the authors understanding of a component. If we only had to opt for one of the established definitions, we would choose the one from Szyperski [2]:

> *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*

Even though this definition does not explicitly mention object orientation (OO), we believe OO is the right choice to realize component-based software, also in the presence of embedded systems. This is why we do not emphasize OO in the remainder of the paper. However, we are aware and agree that CBSE and in particular components can be realized with almost any methodology and programming language.

CBSE Viewpoints and Promises

Over the recent years, conference workshops on CBSE issues were becoming commonplace. Although established definitions for the concept of a component exist (e.g. [2], [3]), many of the workshops still spend considerable time lively discussing their meaning. The meaning in turn influences the expression of opinions with respect to the promises of CBSE and the premises for it to work. For instance, in a summary of the key discussion points from a workshop on CBSE, Brown [4] stated that in spite of the common agreement on the role of a component the interpretation was different. That is, they agreed on the role of a "replacement unit", but depending on one of the following two perspectives the interpretation was rather different. First, the component was seen primarily as a commercial-of-the-shelf (COTS) commodity, so that the emphasis was on reuse issues. Second, the component was seen primarily as a core business asset, and consequently the emphasis was put on abstraction issues. We feel that the divergent opinions on pros and cons of CBSE, on research prioritization, etc. are mainly due to the fact that the person expressing his/her opinion has one particular perspective in the back of his/her head.

In addition to the previously mentioned perspectives, namely COTS based reuse and abstraction of business assets, other perspectives can be found. They seem to stem from phenomena which perhaps coincidentally, but almost concurrently had effect on the software market. First, the urge for application interoperability and distribution fostered middleware technologies such as COM/DCOM or CORBA. Second, the spreading of windows based user interfaces lead to a rapidly growing acceptance of the GUI programming paradigm à la Visual Basic with its related technologies. All this is now also attributed and linked to component-based software and technologies, respectively.

Although not orthogonal, we argue that the below defined four perspectives are sufficient to structure discussions on CBSE and in particular to classify its current promises and premises. Each of the different perspectives sheds a different light on the definition of components, and as a consequence they differ in the prioritization of major technical objectives that shall be achieved by CBSE. The characteristic, objectives, and major issues of the four perspectives are as follows:

(a) *Abstraction perspective*:

At its core, this perspective views CBSE as a natural improvement to object orientation (OO). The prime emphasis is on better abstraction mechanisms, on less domain- orientation, on better separation of concerns, on more explicit object interactions, and on less domain specific solution approaches in favor of more generic architectures (frameworks). Objectives are the factoring out of core business assets, their efficient maintenance, and coping with evolving requirements and

thus evolving systems. The entire software development process and its techniques, methodologies, and notations with special emphasis on the analysis and design phases is considered [5]. Component modeling, identification of candidate components during requirements analysis, and re-factoring of legacy OO applications to component-based applications are areas with ongoing activities.

(b) Interoperability perspective

This perspective is clearly driven by the open system ideas with the emerging interconnection technologies (CORBA, COM) which came in response to several needs. These needs at the same time represent the major objectives: breaking monolithic and tightly coupled systems, and making software work together. I.e., the integration of different applications beyond basic data exchange and across networks must be made less painful. Besides improving the plumbing and wiring technologies, this perspective deals with issues such as opening and wrapping of legacy applications. This shall ultimately result in programmatic access to applications through APIs that follow the programming model which is implied by the chosen component infrastructure (the design and run-time environment for components). The components are typically rather fat, i.e., entire applications or large portions of them. Whether or not the internal structure of these applications is component-based is of less concern.

(c) *Reuse perspective*:

This perspective must be seen under the market context. The establishment of an open component market, where the component is the reuse artifact, is a major issue. The COTS software components shall be tradable units, and therefore be independently deployable and preferably binary units. Thus, the perspective on a component is business oriented. Early supporters of the COTS wave seem to favor this perspective and the works undertaken under the umbrella of COTS (e.g. [6]) is now carried over to component technology. Unsolved issues circle around trusting components [7] (certification, contracts [8]), finding components, standardization of component frameworks for easy deployment, and granularity of components.

(d) *Composition perspective*:

The major objective is the easy composition of an application based on pre-fabricated (hopefully COTS) and self-made components with a minimum of glue and plumbing code exposed to the application programmer. Components are the independent binary units for composition. The already rather mature GUI programming paradigm is the envisioned model, e.g., for graphical server builders. Standardization of component management infrastructures (build- and run-time infrastructures for component-based systems) is important, plug & play composition of entire systems the ultimate goal. Components tend to be small.

Since nowadays any large software development effort encompasses facets of all perspectives and the concept of components is tied to all of them, CBSE must indeed be seen more as a spark than a buzz [1].

Partitioned into the previously characterized perspectives, Table 1 lists the CBSE promises at the highest level of abstraction, i.e., the “marketing” level. They shall later be double-checked against embedded systems software needs.

Abstraction perspective	Interoperability perspective	Reuse perspective	Composition perspective
• *Evolvability.* Improved coping with evolving requirements • *Higher quality software* due to reduced complexity (better separation of concerns, etc.) • *Reduced total life-cycle costs* because of the	• *Application distribution.* Interconnection technologies hide most of the underlying network • *Faster time to market* due to the fact that low-level issues are managed by the chosen	• *Faster time to market* due to employment of COTS components • *Higher quality software* due to employment of well proven and in large quantities distributed COTS components • *Reduced costs* due to at least company wide	• *Programming by non-programmers*. Application programming can be carried out by domain professionals not formally educated in software engineering • *Faster time to market* due to development support through tool

improved evolve-ability	middleware technology • *Single programming model*, steeper learning curve compared with programming with several interaction models and all their idiosyncrasies • *Scalability* with respect to the number of components (and users).	component reuse instead of parallel development or reinvention • Selling individual software components as opposed to entire applications as a *new business model*	automation

Table 1: The different perspectives on components and their promises

In the remainder of the paper we will see which of these perspectives is of particular importance in the area of embedded systems.

Embedded Systems Characteristics

Although the term "embedded systems" is often used, there is no universal definition. For the following discussion, however, a common understanding of the characteristics of such embedded systems is vital. This section therefore lists the main properties of embedded systems as we use the term. Of course, the different properties do not have the same relevance for all the embedded applications. However, it seems to be possible to distinguish between two types of embedded applications as a first approximation: "commodity" applications and "industrial control" applications. Examples of the former are cellular phones, washing machine controllers, or video recorder controllers. The latter group includes a bay control unit in an electrical switchyard, or a chemical plant control system.

Usually, the embedded system is an invisible part of another product (such as a washing machine or a high voltage switchyard) rather than a product itself. For this reason, the product designers try to utilize the cheapest possible hardware solution that handles the job. This leads to *limited hardware* platforms: little memory (in the range of a few kByte to a few MByte), a simple CPU (micro-controller or embedded CPU with integrated peripherals and low energy consumption), no superfluous interfaces, and reduced extensibility. Especially in high volume/low price markets, the hardware costs play an important role. This is unlikely to change in the future, although the hardware prices are shrinking on a monthly basis. Other reasons for limited hardware resources are unwanted heat dissipation and the availability of industrial strength electronic components.

Embedded applications always closely interact with a physical process. Their task is to control or observe a physical process, which might require the embedded application to comply with some timing requirements (*real-time* constraints). Primarily the "industrial control" applications are often *safety critical*. Together with the harsh environments they may have to run in, a wealth of further *industrial requirements* result: electromagnetic compatibility (EMC), industrial environmental requirements (e.g. temperature range), safety certificates, type tests, and so forth. These certificates or type tests apply not only for the hardware, but also for the software, which leads to rather rigid software designs. Another important consequence of the safety critical nature, and/or the expected continuos operation of a plant, is the high *availability* required. Any system failure is expensive (downtime!), if not hazardous.

The "commodity" type of embedded systems is usually installed once and not touched anymore afterwards because they are inaccessible or too inflexible. Since maintenance is difficult or impossible, they have to be *reliable*. Hardware or software upgrades are hardly performed. Some embedded applications have a *long lifetime* (e.g. 20 years), especially in high investment applications.

On the other hand, "industrial control" embedded systems often have to allow for extensions and software upgrades (even on-line), in order to preserve the hardware investments and guarantee continuous operation.

"Commodity" embedded applications are often *autonomous*; the whole functionality has to be implemented on the device. The application is neither able to request services from other devices, nor does it have to provide services to client devices. However, this situation is rapidly changing. More and more "commodity" applications are being integrated in networks where they can interact with other devices.

In contrast, “industrial control” systems often consist of several devices connected to form a *distributed system*. The devices might be specialized for some task, and together implement the complete functionality. Therefore, the embedded devices need to communicate with each other.

Since an embedded system is usually not an appropriate software development platform (e.g. due to lack of resources), software is usually *cross-developed* on a separate development system based on a different technology than the embedded system itself. Finally, the application is linked and installed on the target device. The most popular implementation languages are currently C, Assembler, or graphical languages as defined in IEC 61131. The software development methodology is strongly influenced by the fact that programmers are often application domain specialists rather than software engineers. Since the application often communicates with a physical process on a *low level of abstraction*, e.g. through digital or analog I/O ports, the embedded application is more or less hardware dependent.

Some of the embedded devices – especially “commodity” ones – are *mobile*, which restricts their acceptable energy consumption, weight, size, etc. Mobile devices also require wireless communication, which affects the type of protocols that can be used.

As the above discussion shows, there is no uniform nature of embedded systems. The characteristic property seems to be that embedded system software is not a product itself (like a desktop PC application) but an inseparable part of some other physical product. Properties such as hardware resources, industrial/safety/real-time requirements, lifetime, mobility, or software technologies can vary a lot, depending on the concrete application.

The Case Study: A Platform for Substation Automation

A development project, which was carried out in the years 1995 through 1999, shall serve as our case study. The project team consisted of up to 20 hardware and software engineers who worked at three different locations; i.e. the project had to cope with the difficulties of a truly distributed development. The goal of the project was to develop a new hardware and software platform for substation automation. The application domain ‘substation automation’ comprises control, protection, and monitoring of substations. A substation is a node in a network for transmission of electrical energy. Several products such as a controller, a communication gateway, and different input / output devices originated from this platform.

Another development project that started five years earlier provided a very sound basis for this project. A selection of hardware modules, a fundamental software architecture, a set of implemented software modules, a development process and its methods, a decision on a programming language and the respective development environment, and experienced developers were waiting to be reused. The development process observed the traditional ‘V-model’ with the phases ‘Analysis’, ‘Design’, ‘Implementation’, ‘Integration and Test’ applied to different abstraction levels of the system architecture. ‘Structured Analysis / Real-Time’ according to [9] served as specification method. The design guidelines followed Grady Booch’s ideas about an ‘object-based’ design [10]. The programming language ‘Ada 83’ appeared to be the best choice for safety-critical real-time systems at that time.

System Architecture

Different products implemented on this platform result in the anticipated substation automation system (e.g. a communication gateway, a controller, a bus master – see Figure 1). The products themselves consist of several intelligent modules which have their own processor (in Figure 1 shown for the controller only). Field buses and parallel backplane buses interconnect the modules and products.

Each of these intelligent modules runs software to provide the required functionality. A basic software architecture, which is common to all intelligent modules, eased the development of the software and supported the interoperation between the modules.

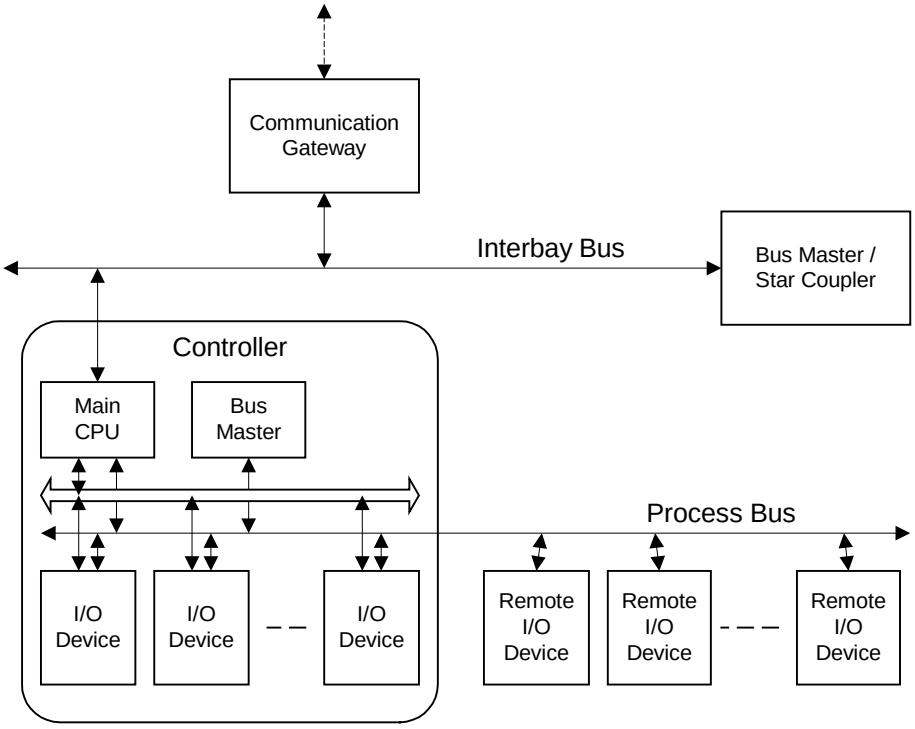

Figure 1: Architecture of the Substation Automation System

Basic Software Architecture

As a consequence of the guidelines adopted from [10], a software architecture arose which decomposed the system into components called 'subsystems'. Figure 2 visualizes this architecture where the three-letter-abbreviations identify the different subsystems. Since their meaning is not relevant to the discussion, we omit their description. The entity 'HW Resources' summarizes several subsystems, which are abstractions of hardware resources (e.g. internal clocks, memories, input / output devices).

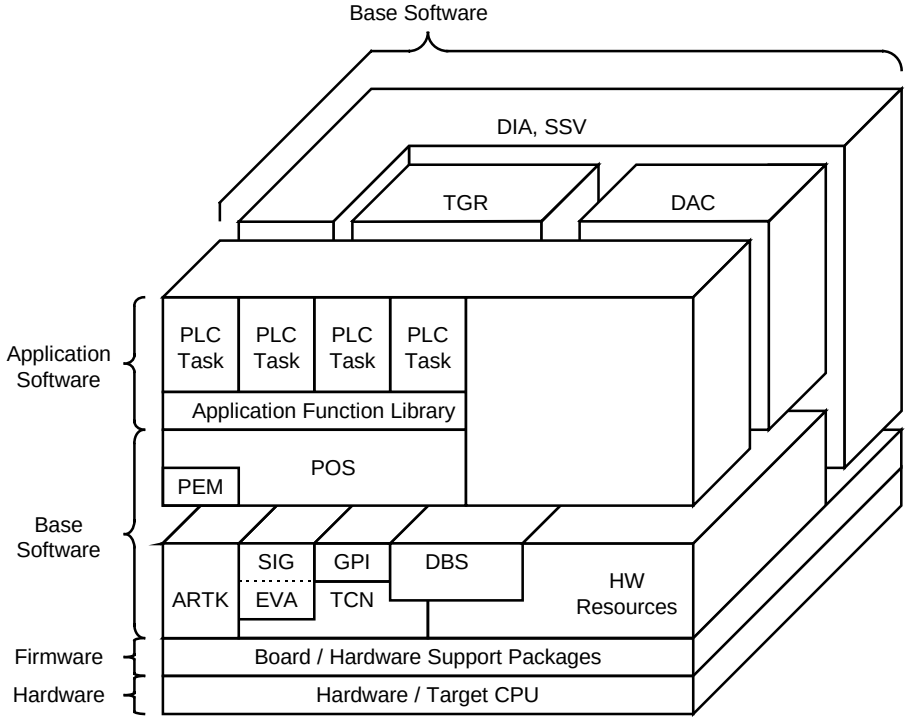

Figure 2: Basic Software Architecture

The subsystems would be candidate software components. Strictly speaking however, in their current form they do not qualify as a software component according to the definition in this paper. As the subsequent section will show, they do, however, emphasize the importance of the first part of Szyperski's definition: *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only ...*"

A generic context description (see Figure 3) specifies the dependencies among the subsystems. Special dependencies exist to the overall system control and diagnosis, to the configuration and process databases, and to clients outside of the system (called 'users'). All subsystems inherit from the root class 'Objects'. The root class provides a system-wide unique identification of all subsystem instances, a state-machine which stores the operation state of the subsystem, and an interface for error messages.

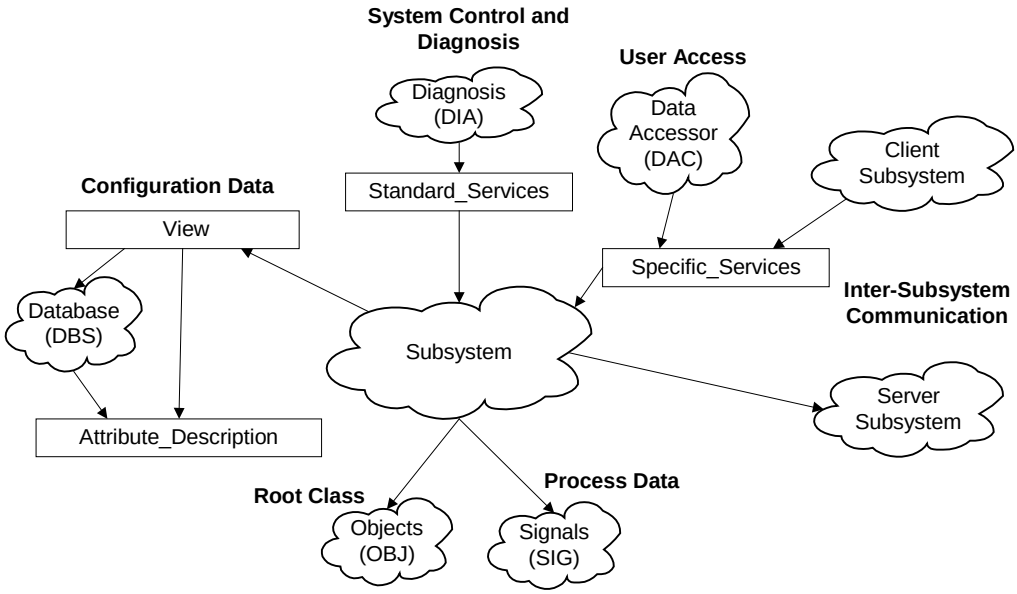

Figure 3: Subsystem Context

Furthermore, all subsystems have well defined interfaces and offer a clearly specified functionality to its users. Each subsystem must offer the following standard services:

Start	This service creates a new instance of the subsystem. The subsystem performs its specific initializations and loads the configuration data from the database subsystem (DBS) through its view builder.
Object_State_Of	The subsystem provides its state to the diagnosis subsystem through this service.
Process	The subsystem performs its function once. This service is used to implement cyclic execution in order to meet hard real-time requirements.
Set_Ready	This service sets the subsystem in a continuous running mode in which the subsystem can perform its functions as often as it is needed for the specific subsystem.
Set_Not_Ready	The subsystem has to stop performing its function.
Shutdown	The subsystem can perform some shutdown activities mainly falling back to a stable quiet state.
Last_Wish	The subsystem is allowed to perform some urgent functions in case of a power fail condition.
Load_New	The subsystem is forced to reload its configuration data from the database subsystem through its view builder.

In addition to these standard services, each subsystem should provide other services, which are specific to its functionality. Hence, the subsystems have specified interfaces as required for a software component but these interfaces are not specified by a contract. There is no formal specification of the interfaces either.

The Ada development and runtime environment requires that the complete software for one target, i.e. one intelligent module, is linked ('bound' in Ada terminology) and downloaded as monolithic block. Independent deployment is therefore not possible for a subsystem. The subsystems vary in size between 2'000 and 50'000 lines of code.

The basic software architecture (as shown in Figure 2) appears in variations on all intelligent modules. The composition of subsystems differs from one intelligent module to the other depending on its functionality. On each module, the 'main program' has the role of the composer. It controls and monitors the execution of all the subsystems on its module by means of the standard services.

Assessment and Lessons Learned[3]

Even if we assume subsystems to be synonymous to components the software architecture described in the case study does not deserve the qualification 'component-based'. Compared with our definition of components, as stated in the 'Introduction' section, it lacks features such as independent deployment and the ability for third-party composition. But we also see weaknesses when it comes to contractually specified interfaces (at least beyond interface signature), and explicit context dependencies ONLY.

The most important lessons we learned are:

1) Several products and modules (re)use the same subsystem. However, the functionality required by the different (re)users is not fully identical. Modifications and add-ons became necessary. With different approaches, we tried to avoid concurrent versions of the same subsystem, with little success however.

2) Some subsystems are highly dependent on others. There are even circular references, i.e. a subsystem references itself via three other subsystems. As a result, reusing these subsystems became very difficult. We had to write stubs for the referenced subsystems in order to satisfy the dependencies.

3) The software architecture does not impose a design approach for the subsystems themselves. The code reviews revealed a broad variety of approaches taken, all flavors from straight forward procedural to object-orientated designs (as much as it was possible with Ada 83 as programming language). The subsystems with a design close to object-orientation did suffer the least from the frequent changes of functionality and are the ones easiest to maintain today.

4) The abstraction of the underlying hardware (real-time clocks, process I/O, …) was not rigid enough at the beginning. Migration of the software from one processor to another within the same family (Motorola 68k) was possible with reasonable effort. The transition to a new processor family (PowerPC) forced us to design and to implement a strict hardware abstraction layer which was a very time consuming task.

5) The subsystem 'POS' (Programmable logic controller Operating System, see Figure 2) contains a third-party software. It is the runtime environment for control applications, which are implemented and downloaded by the users of the controller. The integration was done by writing a wrapper so that this software also became a subsystem with the required interfaces. The only problems with this integration related to task scheduling and to interfacing the hardware (the hardware abstraction layer was not available then).

6) A lot of effort went into the development of the distributed diskless database system (subsystem 'DBS'). This subsystem is a good candidate for the application of a COTS-component. (Too) demanding requirements and a limited availability of real-time databases at that time led to the decision of making instead of buying.

7) To achieve real-time behavior, the relevant parts use strictly cyclical processing. The main program invokes the standard service 'Process' of each subsystem according to a pre-defined

[3] The content of this paragraph is a compilation and interpretation of statements from four software engineers and one of the authors who were involved in this project.

schedule. The subsystems place the parts, which must meet hard real-time requirements into this service. For the parts, which are event-driven or not time-critical, the subsystems may create their own low-priority tasks. The real-time behavior was guaranteed by this approach despite the complexity of the system. However, assigning the right priorities to the low-priority tasks and giving them enough CPU time was a very difficult job and involved a lot of 'try-and-error' loops.

8) The system architecture is highly parallel, i.e. a lot of intelligent modules execute software concurrently (see Figure 1). This concurrency caused a lot of problems, which required a fair amount of synchronization among the subsystems on the different modules.

In summary, the case study showed how complex it is and how much effort is needed to develop a distributed embedded real-time system. The major part of the work went into designing a distributed run-time infrastructure. Once this infrastructure is in place and working, developing the embedded application software itself is less of a problem and comparably inexpensive. Unfortunately, the development costs split is opposite to the value added of the entire product. However, taking an off-the-shelf infrastructure was and is not considered, primarily because of the real-time constraints.

Discussion

Promises

The fact that most of the development effort was dedicated to developing a suitable run-time infrastructure, rather than developing the application logic, should be alarming. Is this typical for embedded systems? We believe it is. If we consider the component-like nature of the infrastructure developed (before mainstream technologies were ready to be considered), we can conclude that the project would have profited from a standard component infrastructure in that the focus could have been put to the *application* development rather than infrastructure development. Furthermore, several areas can be identified where CBSE promises could have contributed to faster development and a better result:

Central seem to be all issues circling around the ideas of *abstraction*, i.e. everything that helps to derive a good architecture (above lessons 1, 2, 3). Hence, the potential benefits of the CBSE abstraction perspective are most promising for our kind of case study and the prime justification to invest into CBSE. Supporting evidence is given through the emphasis on good OO techniques such as:

- Find a good tradeoff between generic, reusable components that require configuration overhead vs. specific, inflexible components that are not well reusable.
- Find a good tradeoff between uniform functionality within components – leading to complex interdependencies among components –, and loosely coupled components with less cohesive internal functionality.

Reuse of COTS components can decrease time-to-market (lessons 5, 6). Components that do not fit into the system architecture can be wrapped. However, it is important to mention that the COTS components reuse perspective depends on the size and quality of a future industrial component market.

Component based *application* development (e.g. graphical *composition*) is promising in the embedded world since it requires much less IT expertise and is therefore suited for domain specialists. For exactly this reason, the case study project integrated the 'POS' execution environment (see lesson 5) that is programmable by a graphical language (IEC 61131). Component infrastructure could provide a programming/runtime environment for system software as well as application software.

This list of potential benefits is certainly not exhaustive. A suitable infrastructure, had it been available, would have raised more desires. For example, after explicitly asking questions such as, "would independent deployment of subsystems have helped in your project?", the software engineers replied with "yes, we would have been the kings...".

Problems/Questions

Thus, a component-like architecture seems to be a promising approach in the design of embedded system, especially based on a standard component infrastructure. But do the well-known technologies used in the PC and server world (DCOM, CORBA, Java) meet the requirements? Some problems and open questions requiring further investigation are:

In case *real-time* performance is required (as in the case study, see lesson 7), how does component infrastructure support real-time? Are the recent initiatives and products such as real-time Java the solution? How do components specify their behavior with respect to real-time (contracts)? *Concurrency* (lesson 8) is also critical: Is plumbing technology available that would support a cyclically synchronous instead of asynchronous communications model?

Interestingly, hardware resources were no big issue in the case study and would therefore not preclude standard component infrastructure. However, this is probably different for other types of embedded systems, particularly for "commodity" ones.

Cross-development of component based applications might pose problems – at least with COM – because the composition tools (running on the development platform) generally use some of the components' runtime functionality (written for the embedded platform). Other technologies such as the Java platform do not suffer this drawback.

An essential point seems to be *hardware independence* (lesson 4). On one hand, this is a question of careful software design (abstraction perspective, see above), in particular for embedded systems due to their hardware related applications. On the other hand, the component infrastructure can support a hardware/platform independent design. Except for the Java platform, component infrastructures do not focus on this issue – some even promote the contrary.

Conclusion

Although it can be argued whether or not this case study is representative for the breed of embedded systems, we feel that after the desktop world, the embedded world could benefit from CBSE as well. To realize this potential, however, embedded systems specific problems have to be solved and suitable component infrastructure to be developed.

References

[1] B. Meyer and C. Mingis, "Component-Based Development: From Buzz to Spark," in *IEEE Computer*, vol. 32, 1999, pp. 35-37.

[2] C. Szyperski, *Component Software - Beyond Object-Oriented Programming*: Addison-Wesley, 1998.

[3] O. Nierstrasz and D. Tsichritzis, *Object-Oriented Software Composition*: Prentice Hall, 1995.

[4] A. W. Brown and K. C. Wallnau, "The Current State of CBSE," in *IEEE Software*, vol. 15, 1998, pp. 37-46.

[5] C. Atkinson, T. Kuehne, and C. Bunse, "Dimensions of Component-based Development," presented at 4th Int'l Workshop on Component-Oriented Programming WCOP'99, Lisbon, Portugal, 1999.

[6] Software Engineering Institute (Carnegie Mellon University), "COTS-Based Systems," http://www.sei.cmu.edu/cbs/overview.html.

[7] B. Meyer et al., "The Trusted Components Initiative," http://trusted-components.org.

[8] A. Beugnard, J.-M. Jezeguel, N. Plouzeau, and D. Watkins, "Making Components Contract Aware," in *IEEE Computer*, vol. 32, 1999.

[9] D. J. Hatley and I. A. Pirbhai, *Strategies for Real-Time System Specification*: Dorset House, 1987.

[10] G. Booch, *Software components with Ada*: Addison-Wesley, 1987.